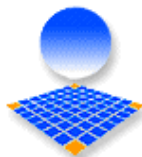


Rapport de projet informatique ULIN 607
de la licence informatique 3^{ème} année
Année 2007/2008

Apprentissage du jeu de morpion

Présenté et soutenu par :
ALMES Romain
BOUADDI Mohamed-Amine
BUENDIA Sandrine
KBIRI-ALAOUI Abdelhakim
LONG Sébastien
SIMARI Julio Ruiz

Dirigé et encadré par :
Mme HUCHARD
Mme NEBUT



Apprentissage du jeu de morpion

Remerciements

Pour la réalisation de ce projet, nous tenons à remercier nos professeurs qui nous ont encadrés tout au long de cette année, et grâce à qui nous avons pu mener à bien le travail qui nous a été confié, depuis l'analyse jusqu'à la programmation.

Dans cette optique, nous remercions également nos tutrices de projet, Mme HUCHARD et Mme NEBUT pour nous avoir encadrés et conseillés tout au long de ce projet.

Sommaire

Glossaire	5
Table des figures	6
Introduction	7
Cahier des charges	8
Description sommaire	8
Spécification technique	8
Analyse du projet	9
Organisation du travail	12
Création d'un jeu de morpion	13
Comment réaliser l'apprentissage	21
Sauvegarde de l'apprentissage	28
Comment décider des coups à jouer	31
Problèmes rencontrés	34
Discussions	35
Conclusion	37
Annexe : manuel d'utilisation	38

Glossaire

Automate : on nomme automate une machine à traiter de l'information, qui se caractérise par différents états, et un système de transition d'états.

IA : Intelligence Artificielle. Construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains, car elles demandent des processus mentaux de haut niveau.

IDE : Integrated, Development Environment. Programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur.

JAVA : Java est une technologie développée par Sun Microsystems. Elle correspond à plusieurs produits et spécifications de logiciels qui, ensembles, constituent un système pour développer et déployer des applications.

SQL : Structured Query Language (que l'on peut traduire par Langage structuré de requêtes), est un langage informatique standard, destiné à interroger ou piloter (modifier contenu et structure) une base de données.

UML : Unified Modeling Language ou « langage de modélisation unifié », est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie de la modélisation objet, utilisée en génie logiciel.

Table des figures

Figure 1 : Diagramme des cas d'utilisation	9
Figure 2 : Diagramme des classes	11
Figure 3 : Diagramme Séquence Interaction	14
Figure 4 : Diagramme des classes simplifié	16
Figure 5 : Interface générale du programme	17
Figure 6 : Interface de jeu.....	18
Figure 7 : Interface du menu option.....	19
Figure 8 : Interface de l'apprentissage rapide	20
Figure 9 : Deux grilles équivalentes dans l'automate	22
Figure 10 : Ajout d'une transition pour éviter la redondance d'état	23
Figure 11 : Explication du mode miroir	27
Figure 12 : Sauvegarde de l'automate dans un fichier	29
Figure 13 : Schéma de la base de données.....	30
Figure 14 : Représentation d'une partie avec la méthode miroir	34
Figure 15 : Représentation de l'algorithme du min-max	36

Introduction

Au cours de la troisième année de Licence Informatique à l'Université Montpellier 2, il est demandé aux étudiants d'effectuer un projet, pour une durée approximative de trois mois. Le but de ce projet est d'offrir aux étudiants un contact avec le travail en groupe, tout en réalisant un projet concret et en respectant les contraintes imposées par le sujet.

Ce projet, encadré par Mme Marianne HUCHARD et Mme Clémentine NEBUT, consiste à réaliser un programme qui est chargé de ne jamais perdre au jeu de morpion. Rappelons que le jeu de morpion se joue à deux joueurs. Chacun joue successivement et le but consiste à aligner trois pions identiques sur une grille de 3*3.

Pour apprendre à jouer, ce programme doit être doté d'une phase d'apprentissage, durant laquelle il va jouer un grand nombre de parties. Pendant cette phase, les parties jouées vont être sauvegardées. Ainsi, une fois l'apprentissage terminé, le programme doit pouvoir jouer au jeu de morpion sans ne jamais perdre de parties, en choisissant dans sa base de connaissances quel coup jouer.

Dans la suite de ce rapport, nous allons vous expliquer les étapes du développement de ce programme, avec tout d'abord, l'analyse du sujet, puis la création d'un jeu de morpion, suivi par la réalisation de la phase d'apprentissage. Nous verrons ensuite la sauvegarde de l'apprentissage, la politique des coups à jouer en fonction de l'apprentissage et, pour finir, les problèmes que nous avons rencontrés.

Cahier des charges

• Description sommaire :

Le projet proposé par nos tutrices est un programme qui sait jouer, sans perdre de parties, au jeu de morpion après avoir effectué une phase d'apprentissage.

Ce projet se décompose en plusieurs parties :

- Créer un jeu de morpion permettant à un joueur humain d'affronter un autre joueur (qu'il soit humain ou ordinateur), ou bien à deux ordinateurs de s'affronter.

- Créer une Intelligence Artificielle qui enregistrera le résultat de toutes les parties jouées dans un automate. Au cours d'une partie, le but de cet IA sera donc de rechercher, dans cet automate, le meilleur moyen pour atteindre, au mieux, la victoire et, au pire, une égalité.

- Créer un système permettant d'enregistrer ce qui a été appris sur le long terme.

• Spécification technique :

Pour réaliser ce projet, nous avons une contrainte, qui était d'utiliser un langage de programmation objet. Le choix étant relativement vaste, nous avons choisi d'utiliser le langage JAVA, puisqu'il s'agit du langage que nous maîtrisons et connaissons le mieux au sein du groupe.

Nous devons également utiliser un automate pour représenter la phase d'apprentissage.

- Outils utilisés :

Le jeu de Morpion sera réalisé avec le langage de programmation JAVA, couplé à une base de données SQL. Pour cela, nous utilisons deux utilitaires :

- Eclipse qui est un IDE que nous connaissons bien, que nous trouvons pratique, complet et performant.

- HSQLDB qui est un système de gestion de base de données relationnelle écrit en JAVA, que nous utilisons pour sa simplicité d'utilisation et ses performances (<http://hsqldb.org>).

Pour ce projet, nous n'avons pas de contraintes concernant la charte graphique. Nous avons donc créé une interface simple visuellement, qui permet de comprendre immédiatement l'intérêt de chaque bouton.

Analyse du projet

Afin d'avoir une idée générale du sujet qui nous a été confié, nous avons commencé par réaliser une analyse comprenant un diagramme des cas d'utilisation, pour nous offrir un aperçu des fonctionnalités à développer, puis un diagramme des classes, pour nous donner une idée de la structure de l'application et des classes à réaliser. Par la suite, nous avons réalisé des diagrammes de séquence afin de nous permettre de décrire graphiquement le déroulement d'une partie.

Tous nos diagrammes ont été réalisés avec power AMC, qui ne respecte malheureusement pas totalement la norme UML, telle la boucle « loop » dans les diagrammes de séquence, ou le cadre représentant le jeu de morpion dans le diagramme des cas d'utilisation.

Ci-après, vous trouverez le diagramme des cas d'utilisation, qui permet d'illustrer les principales fonctions attendues.

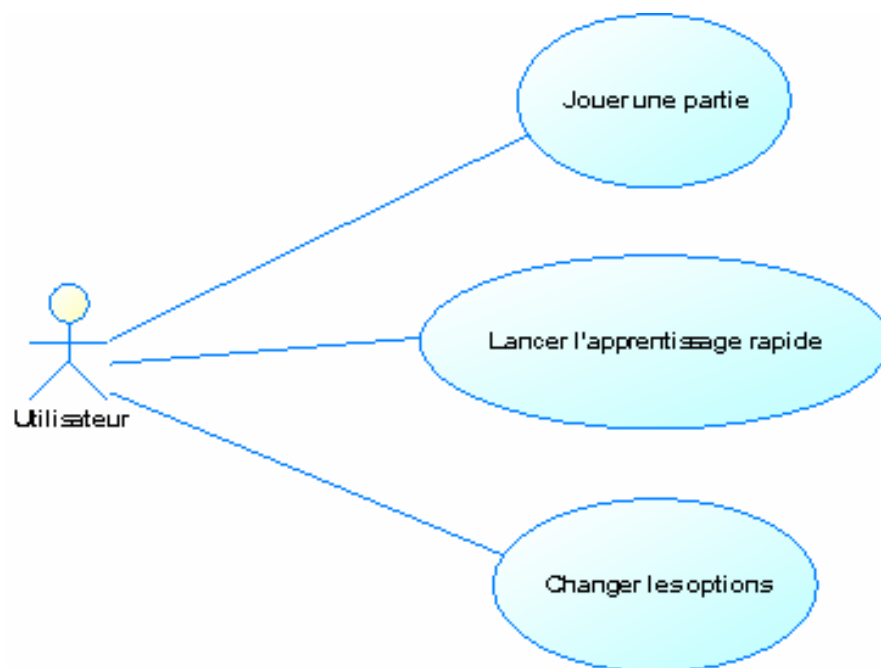


Figure 1 : Diagramme des cas d'utilisation

Ce diagramme des cas d'utilisation montre les types d'interaction que l'application offre à l'utilisateur :

- L'utilisateur peut jouer une partie contre un joueur humain ou l'ordinateur. Il peut également lancer une partie entre deux ordinateurs en modifiant les options du programme.
- L'utilisateur peut donc modifier les options de l'application et ainsi choisir le type de joueurs qui vont s'affronter. Il peut également choisir de remettre à zéro l'apprentissage.
- L'utilisateur peut lancer un apprentissage rapide de l'ordinateur : deux ordinateurs vont s'affronter, l'un contre l'autre, en lançant des parties successives de façon très rapide et ainsi apprendre à jouer.

Pour respecter le cahier des charges et implémenter la phase d'apprentissage, nous avons dû utiliser un automate afin d'enregistrer les coups joués. En effet, un automate nous permet de représenter l'ensemble des coups déjà joués, c'est-à-dire les grilles de jeu qui ont été déjà rencontrées, sous forme d'état. Un état est représenté de la façon suivante :

- Un nom d'état.
- Le numéro du joueur à qui appartient cet état (afin de déterminer si cet état a été rencontré par le joueur 1 ou le joueur 2).
- Une grille.

A ces états s'ajoutent des transitions, qui appartiennent à un état, et qui permettent de déterminer vers quels autres états nous pouvons accéder. Ils sont représentés de la façon suivante :

- Une destination (vers quel état amène cette transition ?),
- Un poids (quelle est la pertinence de jouer ce coup ?).

Ainsi, avec cette méthode, il nous suffit de mettre à jour les poids des transitions en fonction du résultat de la partie. On affecte une valeur positive aux transitions si la partie a été gagnée, sinon une valeur négative. De cette façon, avec la succession des parties, les poids deviennent de plus en plus pertinents et l'ordinateur, en appliquant sa politique de choix des coups à jouer, sait de mieux en mieux jouer.

Nous avons choisi d'implémenter chaque composant de l'automate, à savoir l'état, les transitions, ainsi que l'automate lui-même par des classes.

Nous avons également choisi de ne pas créer de classe grille, pour des raisons d'optimisation et de simplicité, ce qui nous a conduit, par la suite, à l'implémenter sous forme de tableau à 2 dimensions, de taille 3*3 dans la classe « etat ».

En revanche, nous avons opté pour la création d'une classe transition, pour la maniabilité et la clarté du code, sans que cela ait de conséquences sur l'espace mémoire.

A partir de là, nous avons pu détailler un diagramme de classes, que nous avons complété au fur et à mesure du développement du projet, et ceci en raison des nombreuses optimisations que nous avons souhaité apporter au projet.

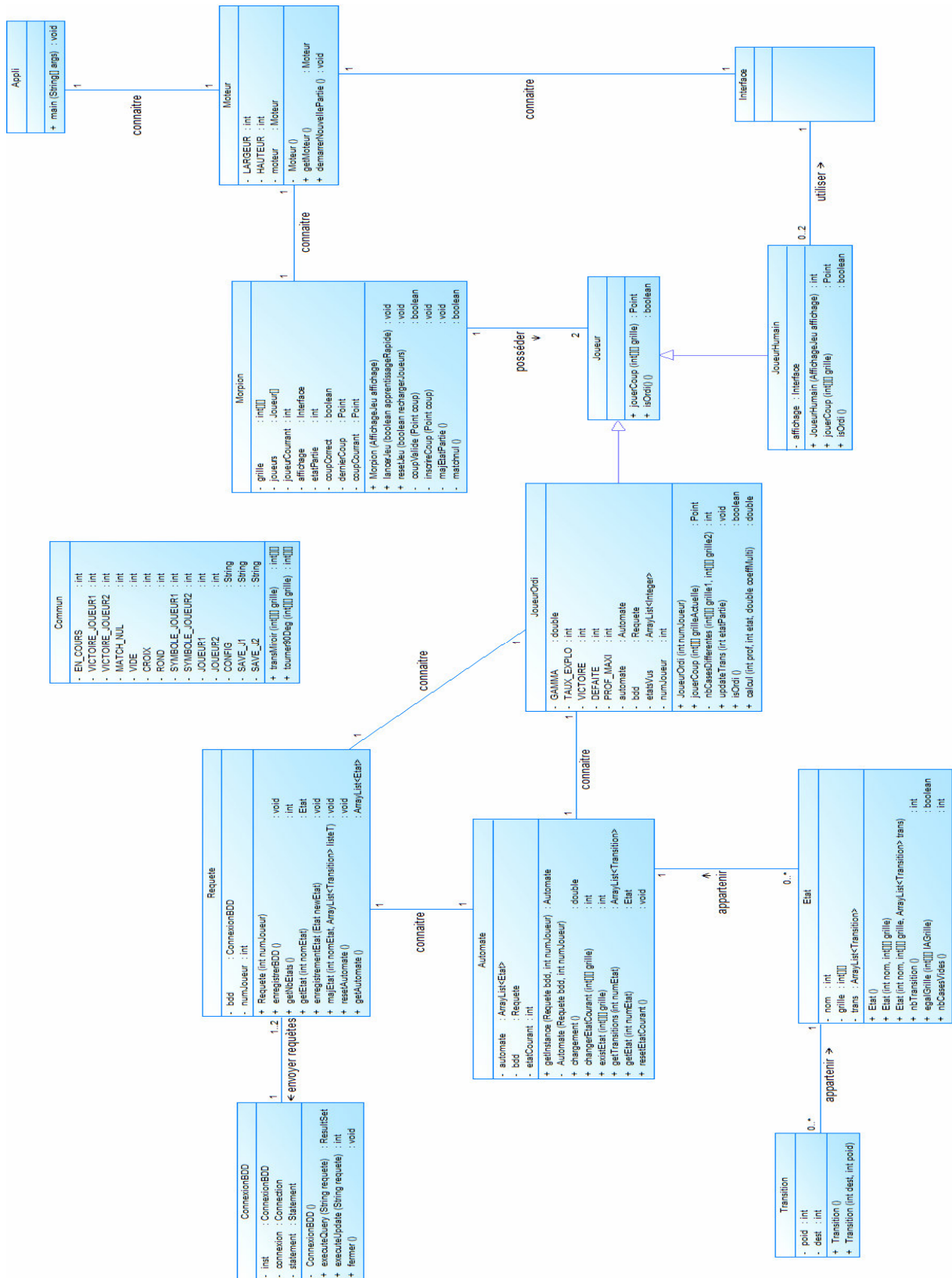


Figure 2 : Diagramme des classes

Organisation du travail

A la suite de cette analyse, nous en sommes arrivés à découper le projet en plusieurs tâches :

- Créer un jeu de morpion.
- Développer la partie automate.
- Gérer l'enregistrement sur disque des données.
- Définir les coups à jouer à partir de l'automate.
- Rédaction du rapport.

Nous avons donc décidé de nous répartir en trois groupes de travail :

- Le premier groupe, composé de Julio RUIZ et Mohamed-Amine BOUADDI, chargé de développer la partie automate.
- Le second groupe, composé de Sébastien LONG et Abdelhakim KBIRI-ALAOUI, chargé de développer le jeu du morpion et la gestion des coups à jouer.
- Le troisième groupe, composé de Romain ALMES et Sandrine BUENDIA, chargé de développer la partie enregistrement des données.

Au cours de ce projet, nous nous sommes réunis de nombreuses fois, afin de discuter de l'avancement de chaque partie, de nous assurer de la compatibilité entre chaque partie et de nous entraider lors des phases de débogage. Pour cela, nous avons utilisé de nombreux moyens de communication (mail, téléphone...).

Création d'un jeu de morpion

Avant de réaliser l'IA du jeu de morpion, nous avons commencé par créer un jeu de morpion, afin de pouvoir tester notre futur programme. Cette partie a donc été réalisée au tout début du projet et ne pris pas beaucoup de temps. En effet, le morpion est un jeu assez simple et très facile à programmer.

Nous verrons tout d'abord comment a été programmé le jeu, puis comment a été conçue l'interface graphique.

- Création du jeu :

Le jeu s'articule autour de la classe Morpion. Celle-ci gère la grille de jeu, le changement de joueurs, et annonce la fin de la partie. Elle connaît donc deux instances de joueurs, qui peuvent être, soit une instance de JoueurHumain, soit une instance de JoueurOrdi.

Après initialisation du jeu, les joueurs commencent à jouer, chacun à leur tour, en appelant la méthode jouerCoup. Celle-ci demande à l'utilisateur de choisir la case à jouer, dans le cas d'un joueur humain, et lance le calcul du coup optimal, dans le cas d'un joueur ordi. Après vérification de la validité du coup, celui-ci est ajouté dans la grille, puis le jeu vérifie si la partie est terminée. Si c'est le cas, un message est envoyé à l'interface pour qu'elle affiche le résultat de la partie (victoire d'un joueur ou match nul), sinon le jeu demande au joueur suivant de jouer.

Pour représenter ce déroulement, nous avons réalisé un Diagramme Séquence Interaction, afin de visualiser les actions entre un joueur humain et un joueur ordinateur.

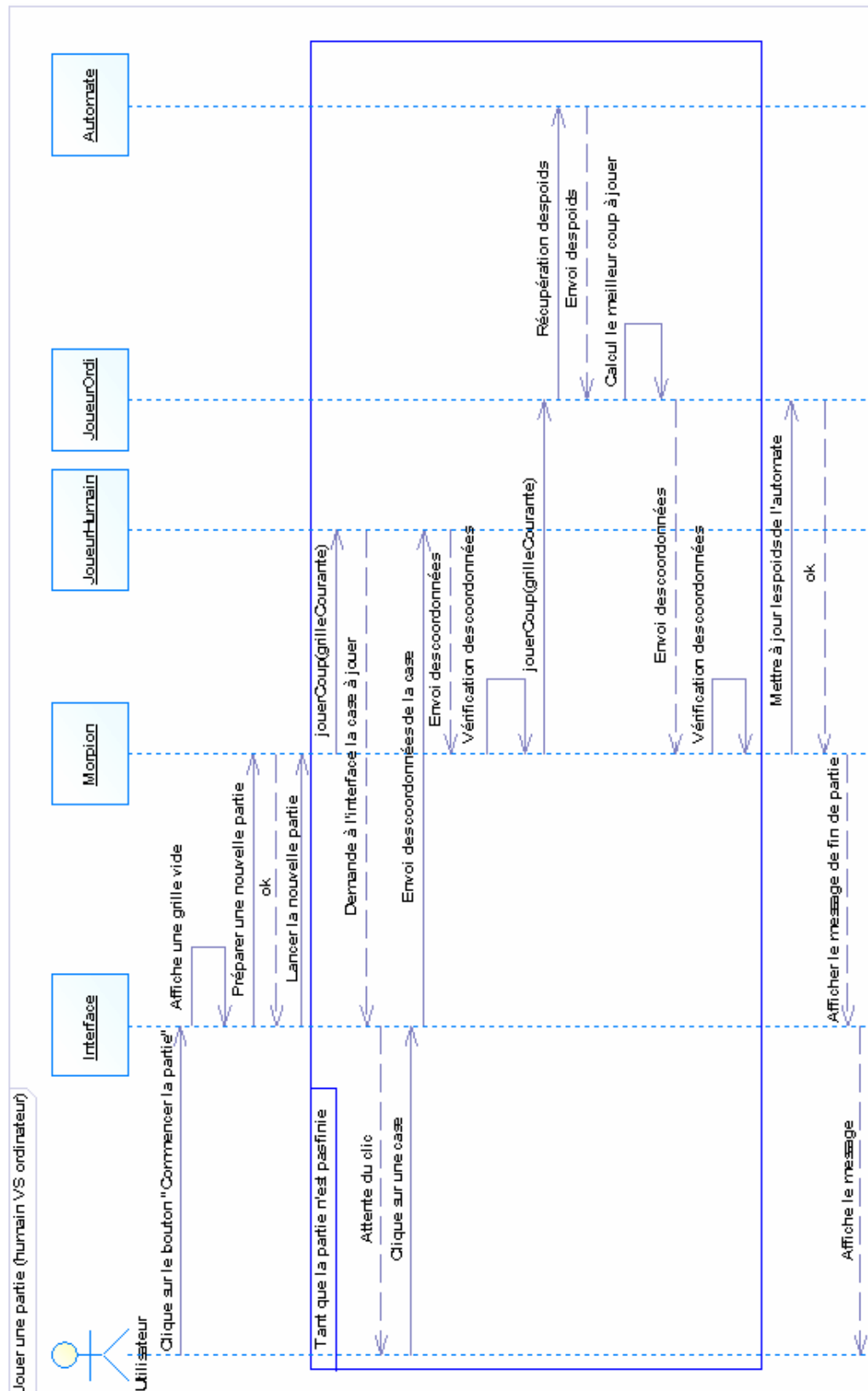


Figure 3 : Diagramme Séquence Interaction

Ce diagramme permet donc de représenter graphiquement le déroulement d'une partie, humain contre ordinateur, dans le cas optimal (les cas critiques et les erreurs de l'utilisateur ne sont pas traités). La partie commence donc lorsque l'utilisateur clique sur le bouton « commencer une partie ». A ce moment-là, nos objets « Interface », « Morpion », « JoueurHumain », « JoueurOrdi » et « Automate » sont déjà créés. L'interface reçoit donc la demande de nouvelle partie, affiche alors une nouvelle grille de jeu vide et demande à l'objet morpion de préparer la nouvelle partie. Une fois la partie préparée, les joueurs jouent un coup, chacun leur tour, jusqu'à ce que celle-ci soit finie.

Jouer un coup pour un humain signifie que le programme demande à l'utilisateur quelle case il souhaite jouer via l'interface, l'utilisateur clique alors dans celle qu'il souhaite jouer.

Jouer un coup pour l'ordinateur signifie appliquer son algorithme de décision à partir des poids retournés par l'automate pour calculer le meilleur coup à jouer.

Dans les deux cas, les coordonnées de la case à jouer sont envoyées à l'objet morpion, qui se charge de les vérifier.

Lorsque la partie est finie, les poids de l'automate sont mis à jour par la classe JoueurOrdi, en mémoire centrale, mais aussi dans la base de données. Le message de fin de partie est alors affiché à l'utilisateur, en lui indiquant s'il a gagné, perdu ou fait match nul.

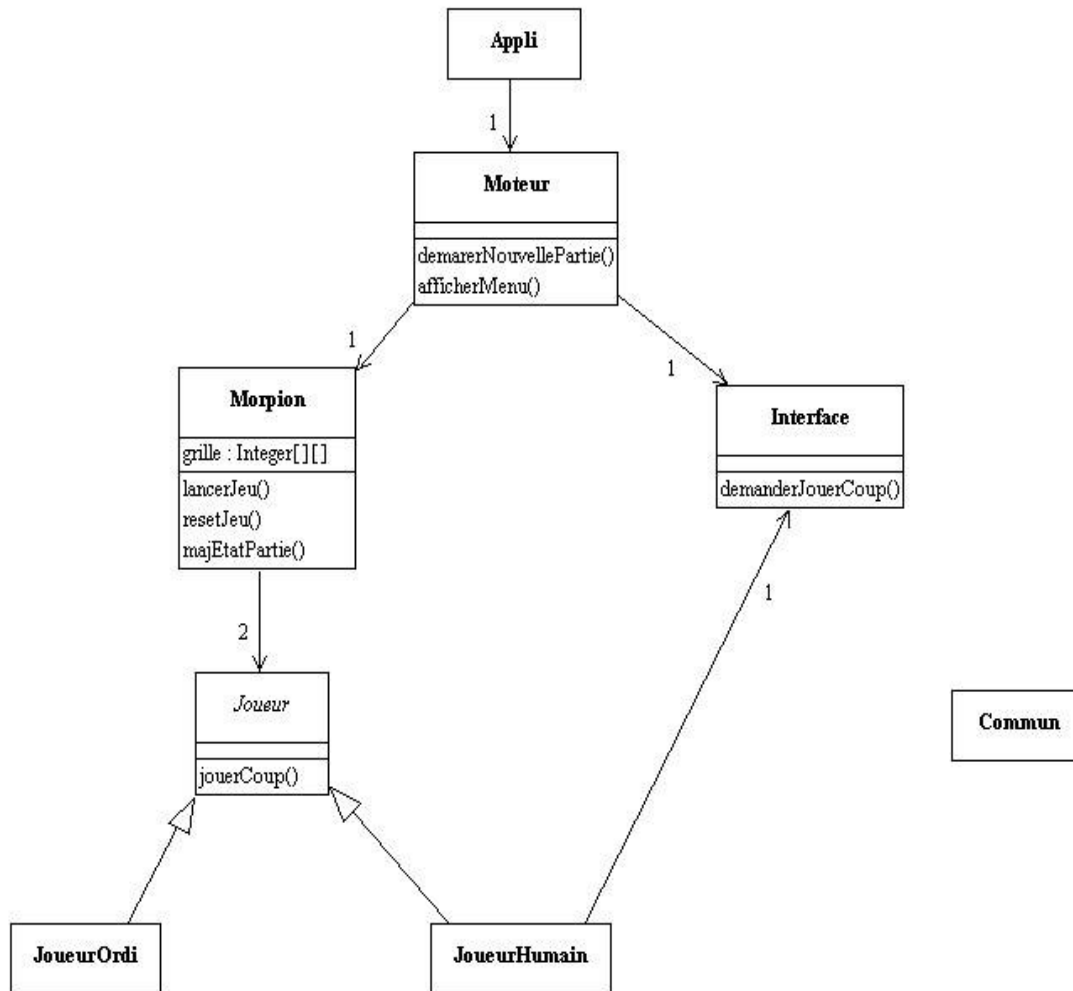


Figure 4 : Diagramme des classes simplifié

Ce diagramme des classes simplifié nous permet de visualiser les classes qui sont utilisées pour réaliser une partie.

- Interface graphique :

Il ne nous était pas demandé de créer une interface graphique pour le jeu, il nous était uniquement demandé un affichage en console afin de pouvoir visualiser le déroulement du jeu. Mais nous avons décidé qu'une interface graphique simpliste augmenterait sensiblement le confort et la facilité d'utilisation du programme, sans ralentir son développement. Nous avons donc opté pour une interface simple mais efficace, composée de gros boutons dont l'utilisation est évidente.

Voici donc la fenêtre principale du programme :

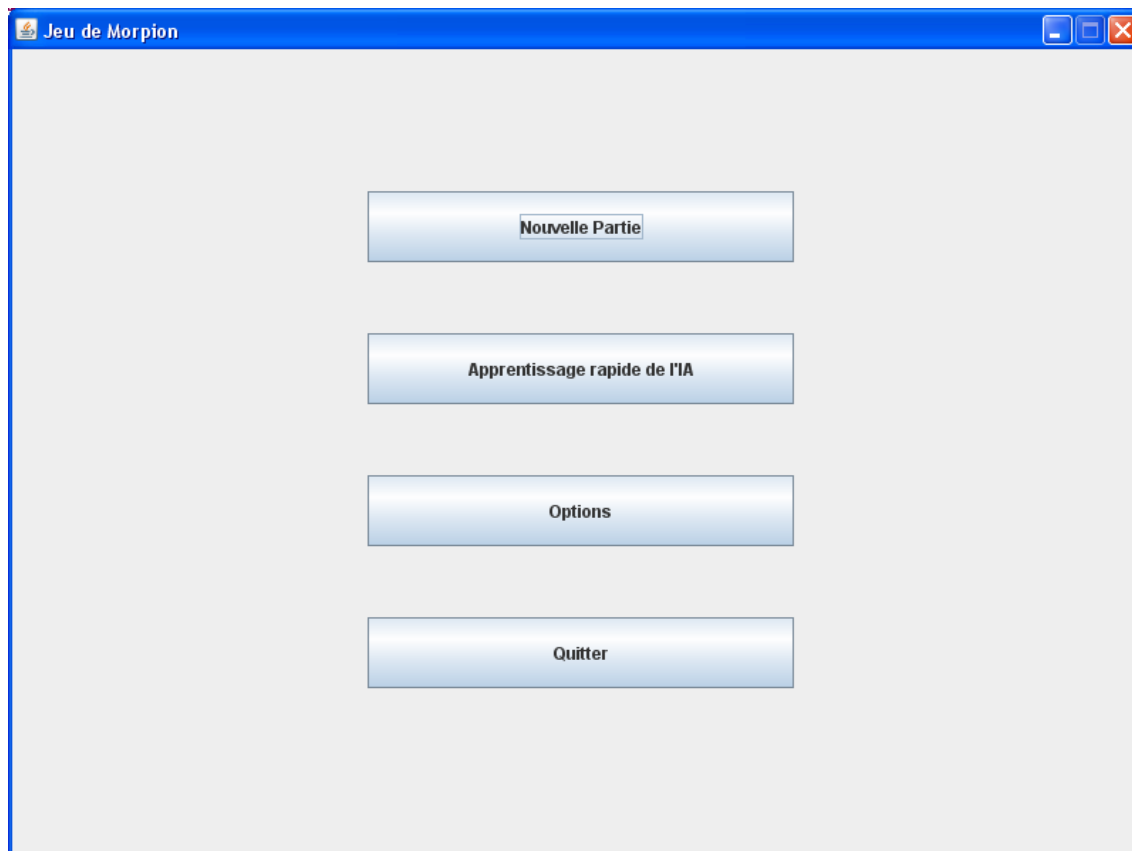


Figure 5 : Interface générale du programme

Celle-ci nous permet de débiter une partie avec les réglages choisis par l'utilisateur, de débiter une phase d'apprentissage rapide de l'Intelligence Artificielle, de changer les réglages du jeu via le menu options, ou de quitter le jeu.

Lorsqu'on clique sur le bouton « nouvelle partie », le programme prépare une grille vide et affiche une nouvelle fenêtre :

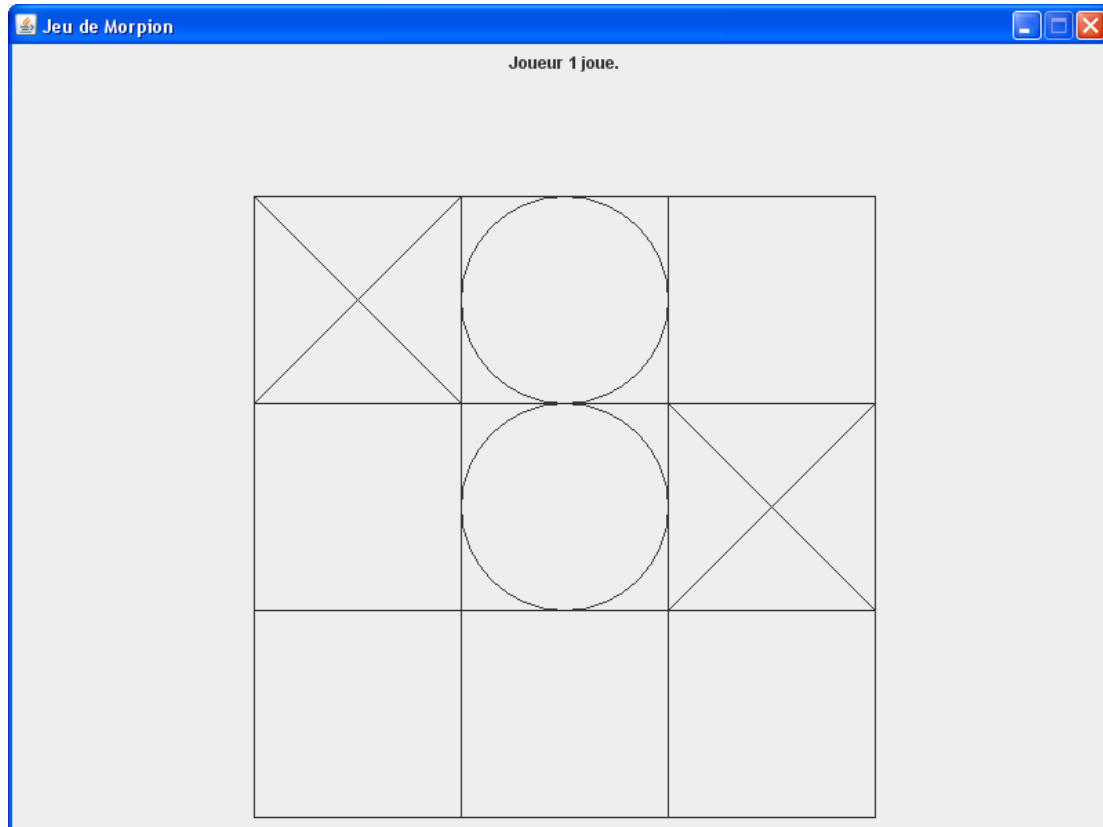


Figure 6 : Interface de jeu

Celle-ci se compose de la grille de jeu actuel, ainsi que d'une barre de message en haut de l'écran qui indique, suivant le cas : le numéro du joueur qui doit jouer le prochain coup, la fin de partie sur la victoire d'un des joueurs ou un match nul. Lorsqu'une partie est terminée, un message apparaît, puis le jeu revient au menu principal.

On peut également régler les options du jeu. En cliquant sur le bouton « Options », une fenêtre permettant de changer les paramètres apparaît :



Figure 7: Interface du menu option

Nous pouvons donc régler le type de chacun des deux joueurs (humain ou ordinateur) ou faire oublier à l'IA les parties qu'elle a déjà vu, et donc recommencer son apprentissage.

L'apprentissage de l'IA étant relativement long, nous pouvons demander au programme de réaliser un apprentissage rapide. Celui-ci commence alors un grand nombre de parties en « ordi Vs ordi » et affiche une fenêtre d'informations qui indique le nombre de parties jouées et la progression de l'apprentissage. Celui-ci peut être interrompu à tout moment en revenant au menu principal, les parties alors jouées seront mémorisées et le programme jouera de mieux en mieux.

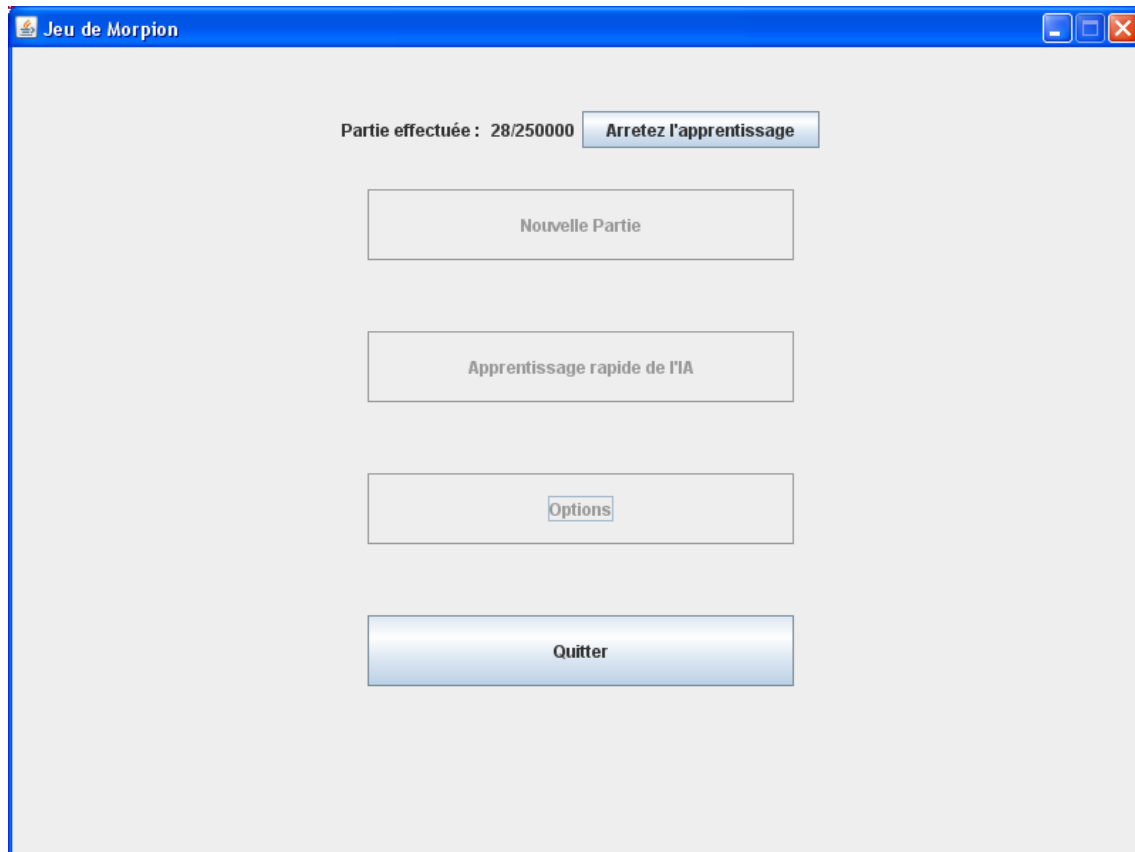


Figure 8: Interface de l'apprentissage rapide

Comment réaliser l'apprentissage

Nous rappelons que, pour enregistrer les parties jouées dans la mémoire centrale, nous avons utilisé un automate où un état correspond à une grille du jeu de morpion et une transition à un coup joué.

Pour concrétiser cette structure, nous avons implémenté chaque composant de l'automate, à savoir un état et une transition, ainsi que l'automate lui-même, par des classes :

- une classe « Transition » qui, après discussions et tests (afin de déterminer la meilleure façon de la représenter), est caractérisée par un poids et une destination. Ceci nous permet d'avoir un code simple et aisément manipulable, sans avoir de conséquences sur l'espace mémoire, à l'inverse de l'utilisation d'un tableau à deux dimensions, comme nous l'avions choisi au départ.

- une classe « Etat » qui décrit l'état d'un automate par son nom, sa grille et sa liste de transitions sortantes.

- une classe « Automate » qui possède comme attributs une liste et une variable représentant le nom de l'état courant de l'automate.

Voyons maintenant comment fonctionne cette partie de notre programme :

Notre programme se sert d'un état particulier, que l'on appelle « état courant » et qui correspond à l'état actuel de la grille.

Après le chargement de l'ensemble des états de l'automate, via le gestionnaire de stockage de données, le fonctionnement de celui-ci repose sur la fonction « changerEtatCourant ». Lors d'une partie, lorsque l'on joue un coup, cette fonction va regarder si le coup joué existe déjà dans l'automate. Elle prend en paramètre une grille et cherche si celle-ci existe. Si c'est le cas, on place ce nouvel état dans l'état courant, puis on retourne sa valeur. Sinon, on crée un nouvel état que l'on ajoute à l'automate et que l'on affecte à l'état courant. Ce fonctionnement est simple mais très coûteux, vu le grand nombre d'états créés par l'application.

Dans un premier temps, nous avons essayé d'exécuter l'application sans optimisations : ainsi, dès qu'un état apparaît dans l'automate, on l'ajoute sans tenir compte du fait qu'il puisse être déjà présent ou pas. Ceci a donc fait apparaître une erreur nous informant que la mémoire vive était saturée. Ce ne fut vraiment pas une surprise, étant donné que, lors notre première réunion, nos tutrices nous avaient prévenu que des problèmes de capacité de mémoire et de saturation apparaîtraient sûrement.

En effet, lors d'une partie, la grille actuelle peut être identique à une autre qui existe déjà dans l'automate, mais sur une branche différente (cf. illustration ci-dessous). Si on ajoute un nouvel état, on observe une redondance de grilles dans l'automate et ceci s'avère très coûteux au niveau de la mémoire.

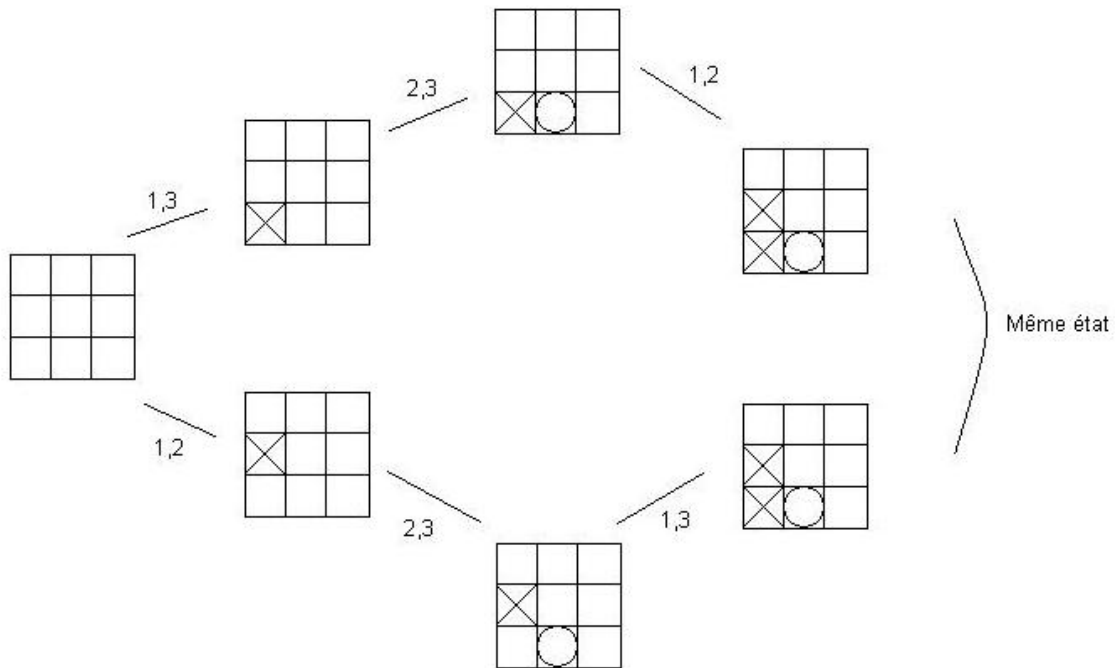


Figure 9 : Deux grilles équivalentes dans l'automate

Pour remédier à ce problème, il nous a suffi d'ajouter une transition de notre état courant vers l'état déjà existant dans l'automate (cf. illustration ci-dessous). Ceci permet d'éviter la saturation de la mémoire, mais ne permet qu'une faible optimisation, car un nombre important d'états est généré par l'application.

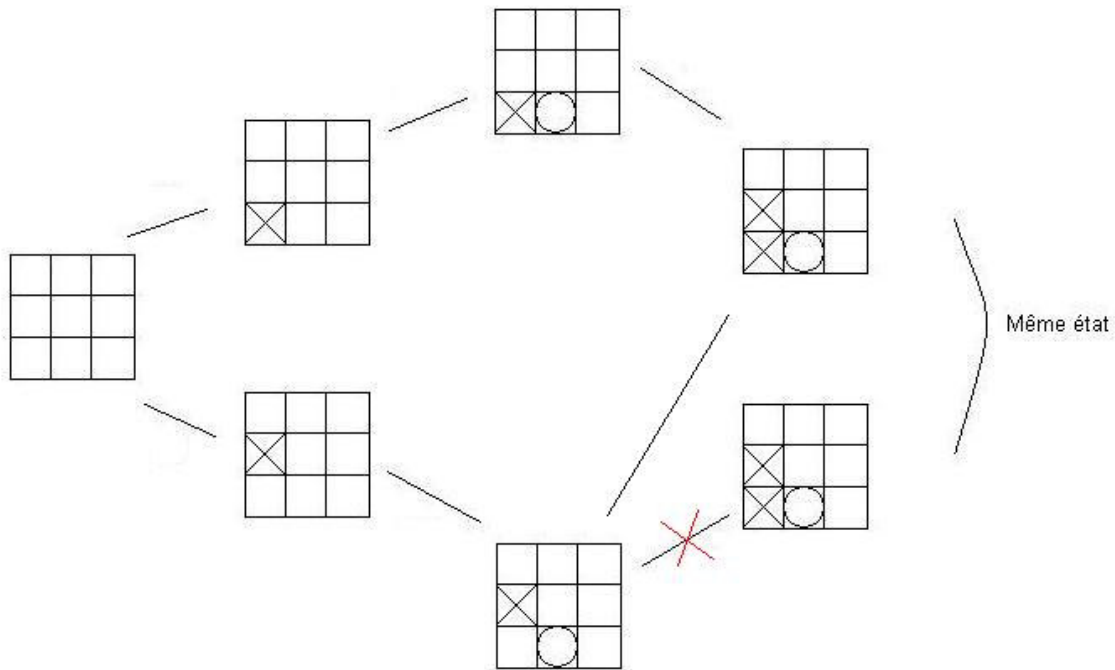


Figure 10 : Ajout d'une transition pour éviter la redondance d'état

Ceci nous a donc poussé à réfléchir sur différentes optimisations que nous pourrions apporter à l'automate, pour réduire le nombre des états, et de ce fait éviter la saturation de la mémoire.

Après réflexion et discussion avec nos tutrices, nous avons proposé deux optimisations différentes que nous pourrions implémenter, afin de réduire considérablement la taille de notre automate.

Voici les deux optimisations proposées :

- La rotation de la grille.
- Le miroir des grilles.

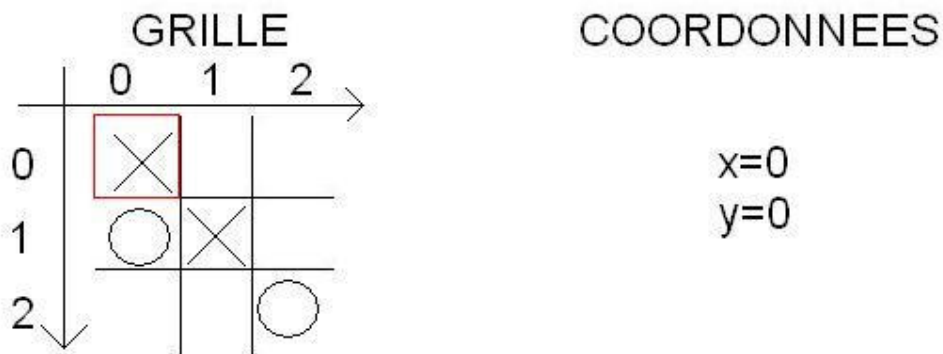
La première optimisation est fonctionnelle et sera détaillée par la suite. La seconde, quant à elle, a été abandonnée à cause des multiples erreurs apparaissant lors de son exécution, mais également parce que nous manquions de temps pour la corriger.

- Rotation de la grille :

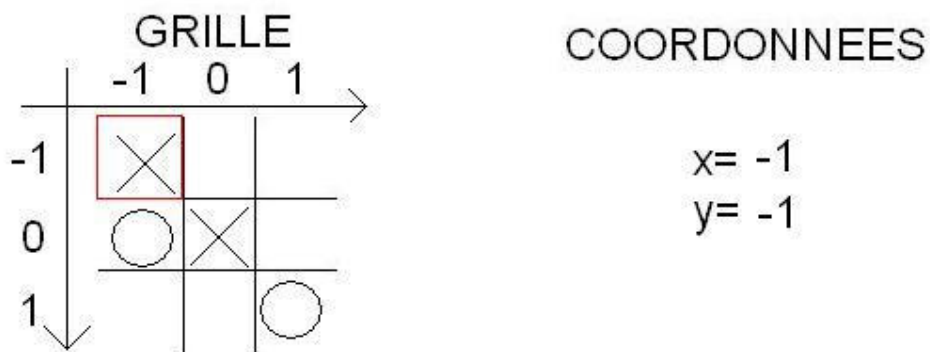
Au début, nous pensions que l'optimisation « rotation de la grille » serait la plus compliquée à réaliser, car nous ne savions pas comment exécuter simplement une rotation d'une grille sur elle-même. Ecrire à la main toutes les coordonnées était bien entendu hors de question, car cette méthode serait source d'erreurs.

C'est alors que nous avons eu l'idée d'utiliser les nombres complexes. En effet, à partir des coordonnées d'un point sous forme de nombre complexe, il est facile d'exécuter une rotation par rapport à l'origine du repère : il suffit de multiplier par « i ». Nous allons donc transformer les coordonnées des points de la grille sous forme de nombre complexe, puis effectuer une rotation sur le nombre obtenu.

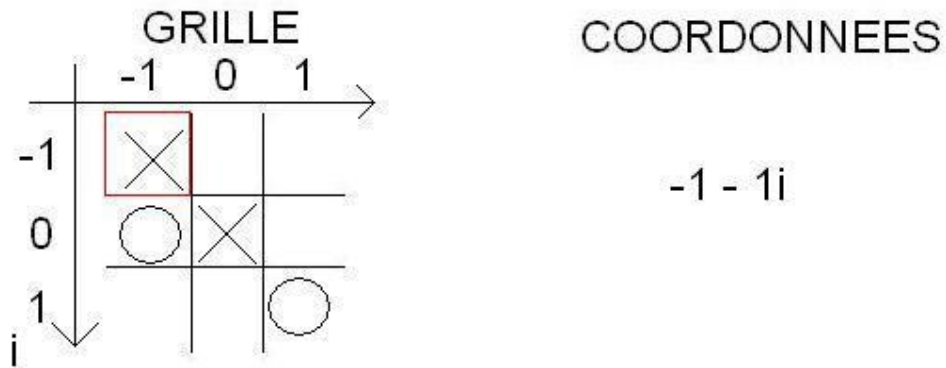
Nous allons voir un exemple en détail. Voici une grille de jeu, les coordonnées sont celles des matrices en java. La case entourée en rouge est celle que l'on va prendre pour exemple. Nous récupérons donc ses coordonnées.



Ensuite nous effectuons un changement de repère, car le centre de rotation de la grille doit se trouver au milieu de celle-ci. Nous avons donc besoin d'avoir la case de coordonnées (0,0) au centre de la grille.



Nous transformons maintenant ces coordonnées en un nombre complexe dont la partie imaginaire représente les ordonnées :

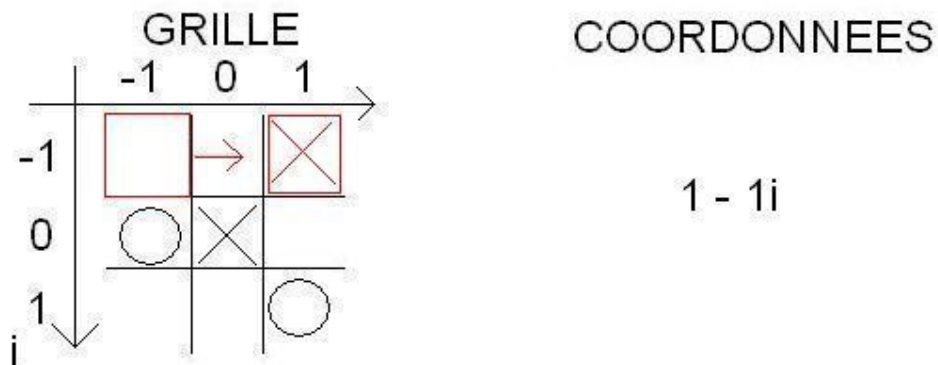


Puis nous effectuons une rotation de 90° , ce qui revient à multiplier le nombre complexe par i .

Rotation de 90° = multiplier par i

$$\begin{aligned}
 (-1 - 1i) * i &= -1i - 1i^2 \\
 &= -1i - (1 * (-1)) \\
 &= -1i + 1 \\
 &= 1 - 1i
 \end{aligned}$$

Puis nous reportons les nouvelles coordonnées sur la grille pour obtenir les coordonnées d'arrivée de la case après la rotation :



La case blanche entourée était la case initiale, la croix rouge entourée est la case après avoir effectué une rotation de 90° sur la droite.

En appliquant ce procédé sur toutes les cases du tableau, on obtient un tableau tourné de 90° sur la droite.

Dans l'implémentation, ces calculs se traduisent tout simplement par :

Coordonnées avant rotation : (X_a, Y_a) , coordonnées après rotation : (X_n, Y_n) .

$$X_n = ((Y_a - 1) * -1) + 1$$

$$Y_n = X_a$$

Explications :

$Y_n = X_a$, car lorsque l'on multiplie le nombre complexe par i , la partie réelle du nombre initial devient la partie imaginaire du nombre final.

$X_n = ((Y_{a-1})^*-1)+1$, car on effectue le changement de repère $(y-1)$, puis on multiplie par i (ce qui revient à multiplier par -1 car la partie imaginaire ne peut être égale qu'à $-i$, 0 ou i), puis on effectue le changement de repère dans l'autre sens afin de récupérer les bonnes coordonnées (le « $+1$ » final).

Ce qui se traduit donc par la fonction suivante :

```
public static int[][] tourner90Deg(int[][] grille)
{
    int[][] res = new int[3][3];

    for(int y=0;y<3;y++)
        for(int x=0;x<3;x++)
            res[((y-1)*-1)+1][x]=grille[x][y];

    return res;
}
```

Cette méthode est statique et se trouve dans la classe « commun », car nous n'avons pas de classe prévue pour gérer la grille et que cette méthode n'avait pas à se trouver dans la classe « automate ».

Ainsi, quand on recherche une grille dans l'automate, on peut également se servir de la rotation pour déterminer si la grille recherchée est présente ou non. Grâce à cela, nous avons pu réduire fortement le nombre d'états à sauvegarder.

- Le miroir de la grille :

Pour un état donné, rien n'est plus ressemblant que son image miroir. C'est-à-dire que, s'il y a un rond dans une case, son image miroir est une croix dans la même case. Il est alors inutile de vouloir stocker les deux états en mémoire, puisqu'il s'agit en fait du même état.

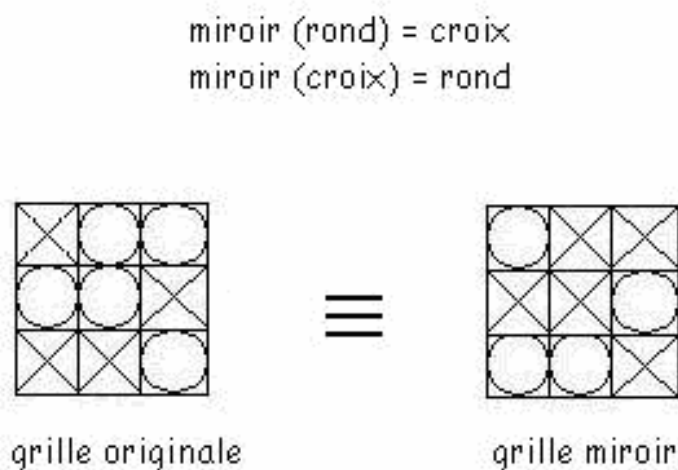


Figure 11 : Explication du mode miroir

Par exemple, si le joueur 1 joue son coup (une croix) dans la case en haut à gauche, mais que cet état n'existe pas dans notre base de connaissances, on va regarder s'il n'existe pas un état miroir à celui-ci, donc avec un rond en haut à gauche.

C'est la seconde optimisation que nous avons proposée, mais nous l'avons désactivée à cause de multiples bogues. Cette partie sera plus développée dans la partie « Problèmes Rencontrés » du rapport.

Pour vérifier que nos optimisations soient vraiment efficaces, nous avons voulu calculer le nombre d'états théoriques que nous aurions dû obtenir. Malheureusement, nous n'avons pas réussi à effectuer ce calcul. En effet, pour le programme avec optimisations (notamment la rotation), nous pensions avoir $9! / 4$ états. Toutefois, ce calcul est inexact, puisque, dans le cas où un coup se trouve au milieu de la grille, la rotation n'engendre aucun changement. Nous ne divisons donc pas notre nombre d'états par quatre.

Nous avons alors pensé à comparer le nombre d'états que nous avons obtenu sans les optimisations, puis le nombre d'états avec les optimisations. Cependant, sans optimisations, au-delà de 20000 états, l'application était vraiment trop lente, nous n'avons donc pas de chiffres plus précis. En ce qui concerne le programme avec les optimisations, nous avons obtenu 1400 états environ.

Sauvegarde de l'apprentissage

Cette partie du programme est destinée à sauvegarder, sur un support de stockage, l'ensemble des états de l'automate.

Nous rappelons que, dans notre programme, lorsque l'ordinateur joue une partie, il met en mémoire tous les coups joués (les états), et en fonction du résultat de la partie, il va attribuer des « poids » aux transitions sortantes, qui sont de valeur positive lorsque le chemin emprunté a permis à l'ordinateur de gagner, et de valeur négative lorsqu'il a perdu.

Il nous faut alors mettre en place un outil nous permettant de sauvegarder ces résultats et d'y accéder facilement.

Les états sont donc fournis par l'automate du programme, puis sauvegardés lorsqu'une partie est terminée. Lorsque l'on commence une nouvelle partie, la liste des états est chargée, puis envoyée à l'automate.

Nous avons alors créé plusieurs systèmes de sauvegarde différents, au fur et à mesure du développement de ce projet, et ceci en raison de nombreux problèmes que nous allons vous expliquer.

- Gestionnaire de fichiers :

Au commencement du projet, nous avons prévu de sauvegarder tous les états rencontrés par l'ordinateur dans un fichier texte, car nous savions comment gérer les entrées-sorties dans un fichier depuis JAVA.

De plus, nous pensions également que le fait d'enregistrer les états dans un fichier texte serait une solution viable, mais après de nombreux tests, il s'est avéré que cette solution était impossible à mettre en œuvre, pour des raisons d'efficacité.

Avec cette méthode, une fois que le fichier contient un nombre d'états assez conséquent, même si le temps de chargement du fichier avant une partie se révèle plutôt rapide, il n'en est pas de même avec la sauvegarde.

A vrai dire, nous pensions qu'en JAVA nous pourrions facilement insérer de nouvelles lignes dans le fichier, ou même modifier les lignes qui correspondraient aux états à mettre à jour.

Or, il n'en est rien. Avec JAVA, il nous faut obligatoirement réécrire l'intégralité du fichier texte contenant les états à la fin de chaque partie, ce qui fait chuter nos performances d'exécution lorsque nous voulons lancer une phase d'apprentissage pour le programme.

En effet, celui-ci met plus de temps à charger et sauvegarder les états, qu'à réaliser une partie.

```
3
0
0;0;0;0;0;0;0;0;0;
1,5;2,0;3,-5;
1
1;0;0;0;0;0;0;0;0;
6,5;7,0;
2
1;0;0;0;-1;0;0;0;
6,0;5,5;
```

Figure 12 : sauvegarde de l'automate dans un fichier

Sur ce schéma, la première ligne correspond aux nombres d'états enregistrés dans le fichier (ici 3).

Ensuite, chaque état est représenté par son numéro (0 pour le premier état), puis par sa grille, codée par « 0 » si la case n'est pas occupée, « 1 » si le joueur 1 a joué dans une case, et « -1 » si c'est le joueur 2 qui a joué dans la case ; et enfin par ses transitions suivies des poids (par exemple, pour l'état « 0 », on a une transition qui va vers l'état « 1 » qui a un poids de « 5 »).

Cette solution, même si elle s'avère fonctionnelle, n'est donc pas des plus optimales.

Après avoir exposé notre problème à nos tutrices, celles-ci nous ont conseillé de trouver une autre façon de gérer le problème des fichiers ou, si nous n'obtenions aucun résultat, de voir avec les bases de données.

Nous avons alors songé à stocker chaque état dans un fichier texte séparé, mais nous nous sommes rendu compte, après avoir fait un certain nombre de tests, que les temps d'accès à chaque fichier étaient vraiment trop élevés.

Ainsi, même si nous évitions la contrainte de tout réécrire lors de la sauvegarde de l'automate, les temps d'accès pour charger tous les fichiers de l'automate rendaient cette solution non viable.

- Base de données

Pour remédier à ce problème, nous avons alors pensé à utiliser une base de données SQL.

Voici le schéma de notre base de données :

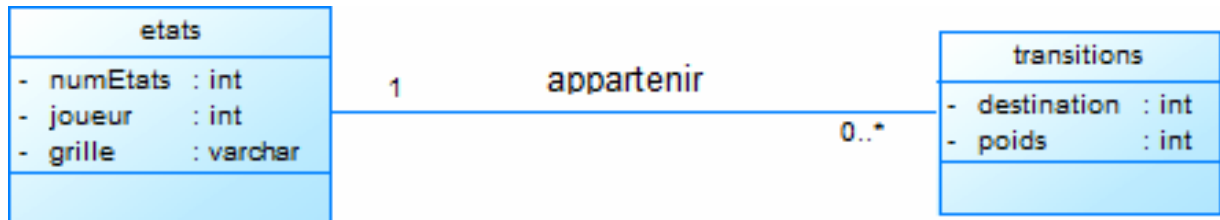


Figure 13 : Schéma de la base de données

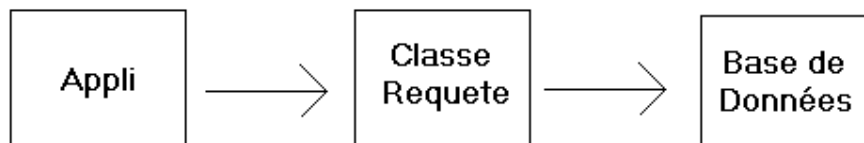
On peut donc voir qu'un état est composé d'un numéro d'état, d'un numéro de joueur à qui il appartient et d'une grille codée sous forme de chaîne de caractères. Il peut posséder plusieurs transitions, mais une transition n'appartient qu'à un état.

Une transition est composée d'une destination, c'est-à-dire un état vers lequel elle pointe, et d'un poids qui correspond à la pertinence de ce coup.

Grâce à cette base de données, nous avons pu résoudre le problème de mise à jour des états, puisque nous pouvons les mettre à jour de façon individuelle.

Au début, nous exécutions beaucoup de petites requêtes pour récupérer les informations, mais nous nous sommes rendu compte que cela ralentissait l'application. Nous avons alors préféré réécrire les requêtes de façon à ce qu'elles récupèrent plus d'informations d'un coup, et ainsi les traiter directement dans le programme.

Afin d'éviter d'avoir des requêtes à l'intérieur de notre code JAVA, nous avons choisi de faire notre application en 2 couches, avec la couche application qui communique avec la couche de requêtes, et c'est celle-ci qui communiquera avec la base de données.



Comment décider du meilleur coup à jouer

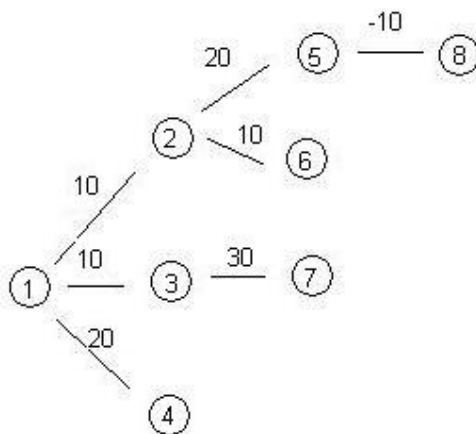
Le but principal du programme est de réussir à décider quel est le meilleur coup à jouer à chaque instant. Pour cela, nous avons mis en place un automate qui mémorise les parties déjà vues et un gestionnaire de sauvegarde qui permet de conserver ces données sur le long terme. Cette partie expliquera comment, à partir d'un automate déjà construit, le programme décide des coordonnées à jouer.

Lorsque c'est au tour de l'ordinateur de jouer, le programme appelle la méthode « jouerCoup » de la classe JoueurOrdi. Celle-ci commence par informer l'automate quel coup a joué l'autre joueur, afin de mettre à jour l'état courant.

On récupère ensuite les transitions sortantes de l'état courant et nous regardons les poids. Pour chaque transition, on effectue le calcul suivant :

$\text{poidFinal} = \text{poids de la transition} + 0.5 * \text{poids maximum des transitions suivantes} + 0.5^2 * \text{poids maximum des transitions d'après, etc...}$ ceci répété 9 fois.

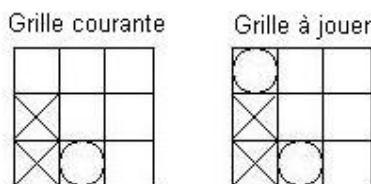
Ce schéma illustre le fonctionnement :



Dans cet exemple, le poids de la transition 1->2 est : $10 + 20 * 0.5 + (-10) * 0.5^2 = 17.5$, celui de la transition 1->3 : $10 + 30 * 0.5 = 25$ et celui de la transition 1->4 : 20.

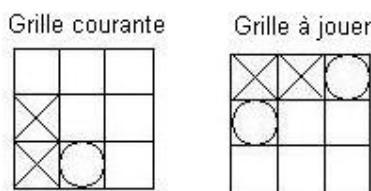
Ensuite, nous choisissons la transition qui obtient le meilleur score et recoupons sa grille, afin de l'analyser et rechercher le dernier coup joué. La fonction renvoie ensuite les coordonnées du coup à jouer au programme principal, après avoir informé l'automate du coup qu'il a choisi.

L'analyse de la grille à jouer a provoqué le plus gros bogue du programme. Celui-ci était dû aux optimisations de l'automate. Avant les optimisations, il n'y avait qu'une case de différence entre la grille courante et la grille à jouer :



Pour l'analyser, afin de trouver le dernier coup, nous avons donc seulement à parcourir la grille à la recherche de la première case différente.

Le problème est apparu lorsqu'on a optimisé l'automate. La grille à jouer pouvait alors être totalement différente de la grille actuelle, car tournée à 90° ou en miroir :



Le parcours de la grille qui permet de rechercher la différence de case renvoyait alors toujours la case de coordonnées (1,1).

Là où tout se compliquait, c'est que le programme pensait que le coup était correct, et informait l'automate qu'il allait jouer dans cette position-là. L'automate changeait alors l'état courant, mais le coup était refusé par le programme principal, car non valide (en général la case (1,1) était déjà prise). Le programme continuait donc son exécution avec un mauvais état courant et rajoutait des transitions aux mauvais états. Ceci provoquait des boucles dans l'automate.

La correction de ce bogue fut longue et difficile, car il se manifestait par des boucles dans l'automate, mais provenait d'une toute autre fonction. Nous avons donc cherché un long moment un bogue dans une classe où il n'y en avait pas. Au final, seuls des tests avec différentes optimisations (miroir et grille tournée sur elle-même) nous ont fait comprendre que le bogue ne venait pas de l'automate mais d'ailleurs. Quelques heures (et des dizaines de fichiers logs) plus tard, nous localisons le bogue et le corrigeons. Pour ce faire, nous avons juste changé la méthode d'analyse de la grille qui va alors compter le nombre de cases différentes et effectuer des transformations sur la grille actuelle (rotations et miroir) jusqu'à ce qu'il ne reste plus qu'une case de différence avec la grille à jouer, puis le parcours de la grille peut se dérouler normalement.

Toutefois, si le programme choisissait toujours la transition de poids maximum pour jouer, il n'explorerait jamais certaines branches de l'automate qui pourraient se révéler victorieuses. C'est pourquoi nos tutrices nous ont demandé de mettre en place un système de pourcentage qui va décider d'aller explorer une autre branche (par exemple, 5% de chance d'explorer une nouvelle branche).

Au début, ce système vérifiait si l'on connaissait toutes les transitions possibles à partir de l'état courant et, si c'était le cas, ne lançait pas l'exploration. Ceci se révéla inefficace, car le programme explorait très peu d'autre branche. Nous avons donc enlevé cette condition et laissé uniquement une probabilité d'explorer, au hasard, n'importe quelle autre branche (déjà connue ou non). Après plusieurs tests, nous avons choisi de régler cette probabilité à 5% en temps normal et à 60% pendant l'apprentissage rapide.

Problèmes rencontrés

Le premier problème a été celui de la performance du programme. En effet, dans la première version de celui-ci, nous nous sommes rendus compte que, lors de l'exécution et du lancement de l'apprentissage rapide du morpion, celui-ci ne pouvait pas dépasser un certain nombre de parties, à cause de la place que prenait l'automate en mémoire.

Effectivement, certains états identiques étaient présents plusieurs fois dans l'automate et le rendaient donc lourd, ce qui saturait la mémoire. Il a donc fallu envisager certaines optimisations, afin d'identifier les états redondants et, par conséquent, alléger l'automate.

Ensuite, un autre problème que nous n'avons pas pu résoudre par manque de temps et qui découle du problème précédent, fut celui du mode miroir. En fait, lorsque nous avons réalisé la partie de notre programme qui gère l'automate et que nous avons voulu l'optimiser pour des raisons de performances, nous avons dans l'idée de construire une méthode « miroir » qui nous aurait permis de diviser par deux notre nombre d'états.

Voyons pourquoi cette méthode n'a pas fonctionné :

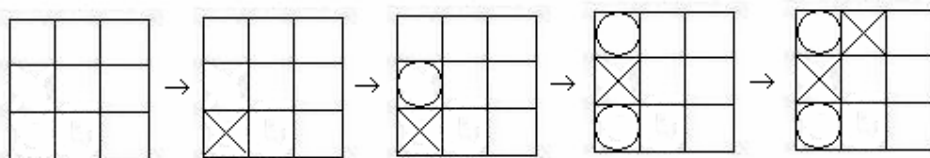


Figure 14 : Représentation d'une partie avec la méthode miroir

Lors d'une partie, on voit que le joueur 1 commence par jouer en plaçant une croix (grille 2). Le joueur 2 joue alors son coup en plaçant un rond (grille 3). Au tour suivant, on se rend compte que l'état que veut jouer l'ordinateur existe en mode miroir, il va donc être inversé (grille 4). On va alors continuer la partie ainsi.

Une fois la partie finie, il faudrait gérer la mise à jour des poids différemment entre les grilles « normales » et « miroir », puisqu'il ne faut pas mettre des poids positifs sur toute la branche de l'automate. Par exemple, si les croix gagnent et que l'on a utilisé la méthode miroir, on va mettre des poids de valeurs positives sur toutes les croix, alors qu'au départ le joueur était un rond. Donc, le joueur qui a perdu (les ronds), possédait les croix au début de la partie. Ses transitions vont donc obtenir des poids de valeur positive, alors qu'il a perdu.

Discussion

Nous regrettons qu'à la fin de notre projet, nous n'ayons pas pu fournir une version du programme avec toutes les fonctionnalités que nous avions prévues, et notamment la méthode « miroir ».

Cependant, nous sommes tout de même heureux d'avoir mené à bien ce projet et d'avoir créé un programme capable d'apprendre à jouer au jeu de morpion.

Dans le cadre de ce projet, nous aurions également aimé pouvoir tester notre programme sur une grille plus grande, dans le style d'un jeu de puissance 4, afin de pouvoir étudier l'évolution de l'automate, ainsi que ses performances.

Cela aurait pu également nous permettre de produire un code plus portable : en effet, notre programme pourrait s'adapter à plusieurs jeux du même genre, uniquement en changeant les paramètres de la grille de jeu.

Nous aurions également voulu implémenter l'arrêt automatique de l'apprentissage rapide : en effet, celui-ci aurait pu s'arrêter de lui-même lorsqu'il aurait atteint une certaine stabilité et qu'il lui aurait été inutile de jouer plus de parties.

Toutefois, nous sommes ravis de cette expérience de travail en groupe, qui est loin d'être aisée.

Nous avons également trouvé intéressant de comparer ce système d'intelligence artificielle à d'autres systèmes existants, et tout particulièrement à l'algorithme « Min-Max » qui lui ressemble énormément et que l'un d'entre nous avait eu l'opportunité de programmer dans un précédent projet.

Cet algorithme fonctionne sur le principe suivant :

- Il simule des coups en avance et crée un arbre.
- Il calcule une note pour chaque feuille de l'arbre.
- Il fait remonter les valeurs à la racine pour trouver le meilleur coup.

Chaque nœud de l'arbre correspond donc à un état de la partie. On alterne entre les nœuds « min » et les nœuds « max ». Un nœud appelé « min » représente le coup de l'adversaire, car on suppose que l'adversaire joue bien et donc choisit le coup qui l'avantage le plus (donc le plus mauvais coup pour l'IA). Ce nœud prend donc la valeur minimale de ses fils et donc est appelé nœud « min ». De même, à l'inverse, pour les nœuds « max ».

Voici un exemple qui illustre son fonctionnement :

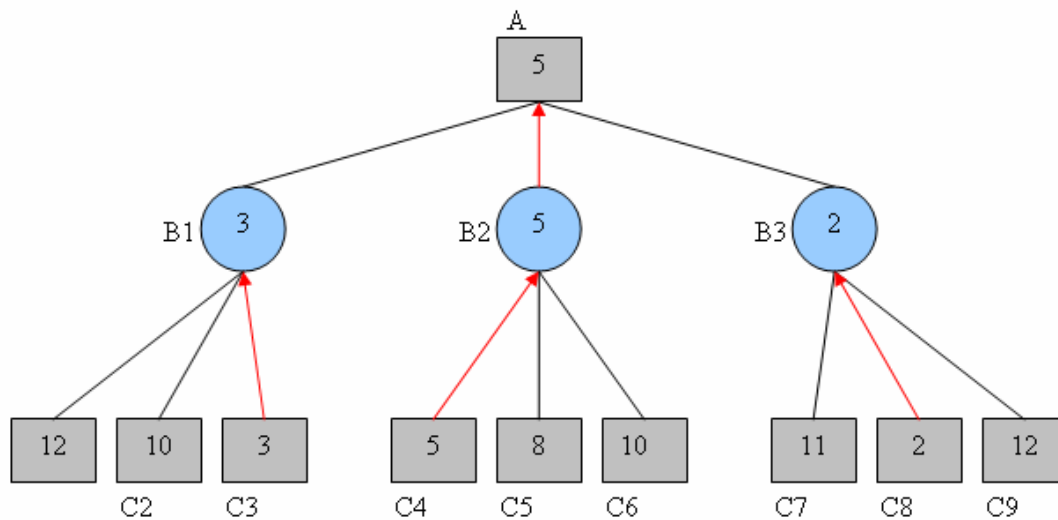


Figure 15 : Représentation de l'algorithme du min-max

Dans ce schéma, les carrés correspondent aux nœuds max et les ronds aux nœuds min. Les états C1-C9 sont les feuilles de l'arbre et leurs notes sont calculées par une fonction d'évaluation. Ensuite, les notes sont remontées jusqu'à la racine, les nœuds B1-B3 sont des nœuds min. Ils prennent donc comme valeur la plus petite de leurs fils, le nœud A est un nœud max, il prend donc comme valeur la plus grande de ses fils. Ainsi, nous avons calculé que le chemin qui a le plus de chance de gagner, est celui qui commence par A, passe par B2, puis par C4.

Ce principe ressemble énormément à notre automate, mais calcule les poids à chaque exécution, alors que l'automate les met à jour, au fur et à mesure des parties.

L'avantage du min-max est qu'il nécessite peu d'espace mémoire, car il ne conserve aucune trace des parties qu'il a déjà vues. Un autre avantage est qu'il sait toujours bien jouer et ne nécessite pas de période d'apprentissage. En contrepartie, il perd en temps d'exécution, car il recalcule l'intégralité des poids à chaque fois, alors que notre automate a seulement besoin de lire les poids sauvegardés en mémoire.

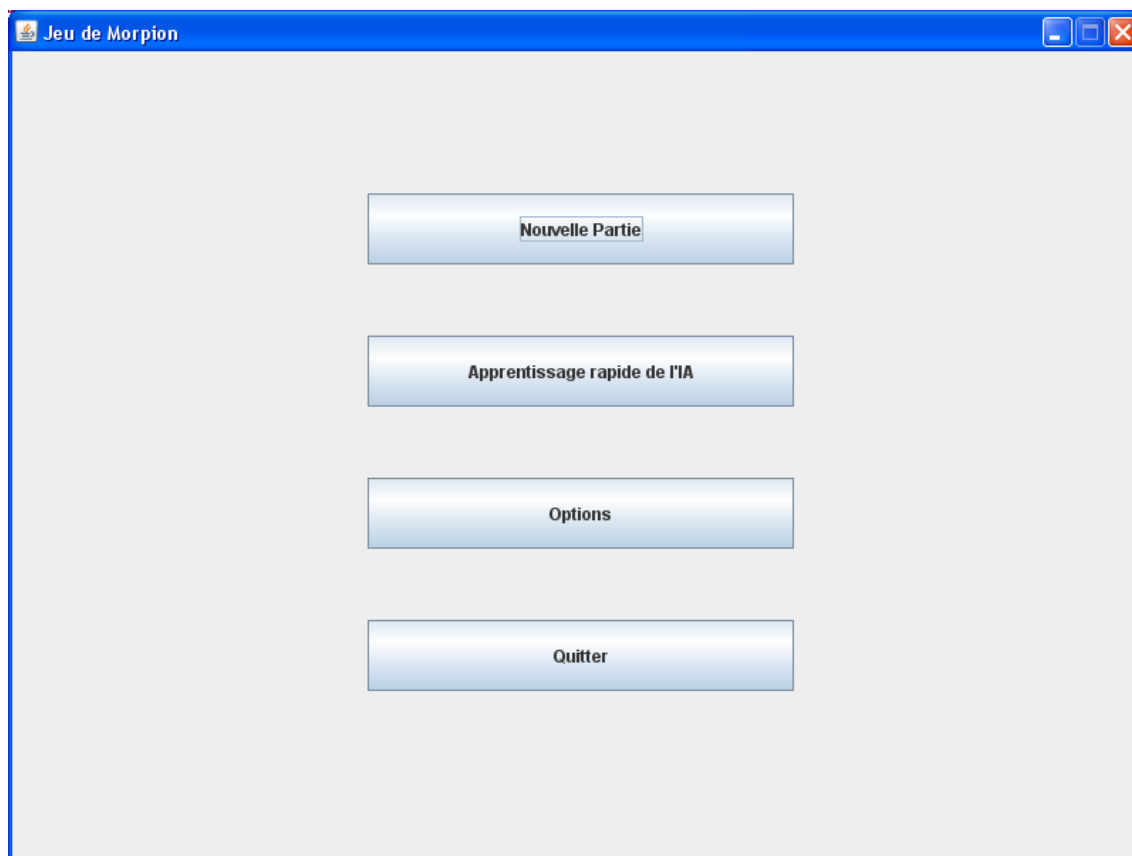
Conclusion

Au cours de ce projet, nous avons réalisé un jeu de morpion et nous pouvons dire que nous avons atteint nos objectifs initiaux. Notre programme est capable d'apprendre à jouer au morpion, pour finalement arriver à ne plus perdre de partie.

En effet, nous avons bien développé une partie capable de réaliser un apprentissage, une autre capable de le sauvegarder, puis une dernière qui sait utiliser cette sauvegarde, tout en respectant les exigences qui nous avaient été données.

D'un point de vue individuel, cette expérience a été vraiment très enrichissante, que cela soit au niveau du travail en groupe ou au niveau de l'apprentissage du langage JAVA. Il est vrai que le travail en groupe nous a permis de mieux nous préparer pour notre vie professionnelle, puisqu'il nous a fallu apprendre à respecter des délais, à communiquer. Heureusement, nous n'avons pas eu de difficultés particulières en ce qui concerne la synchronisation des équipes. En résumé, nous avons appris à nous organiser.

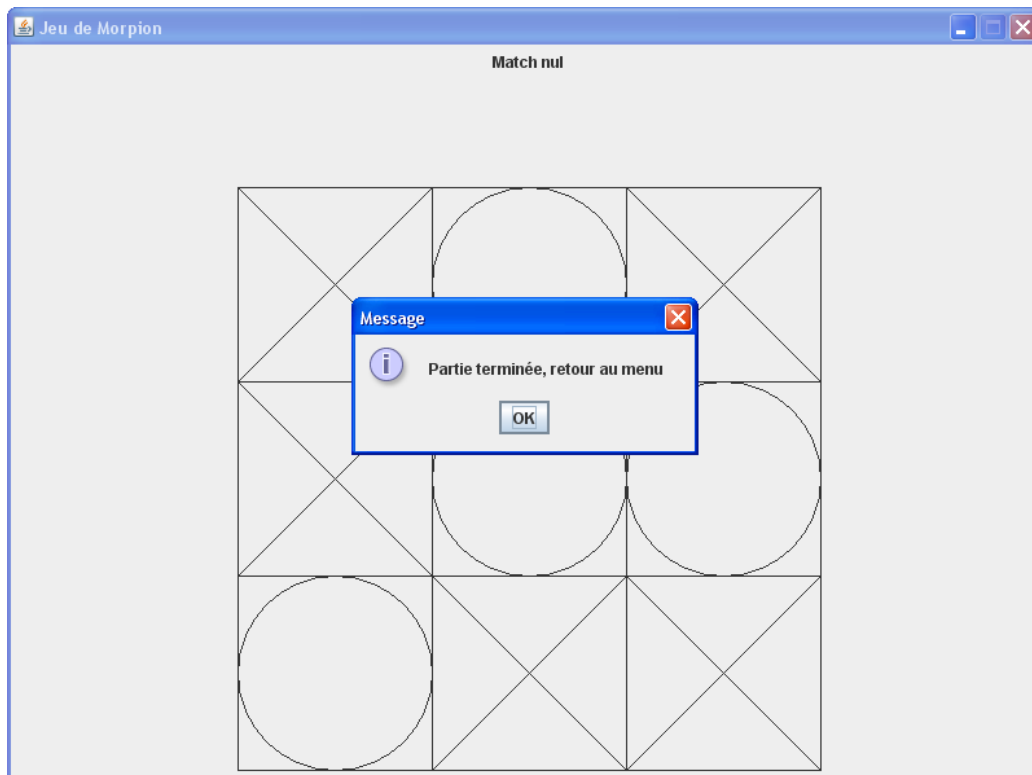
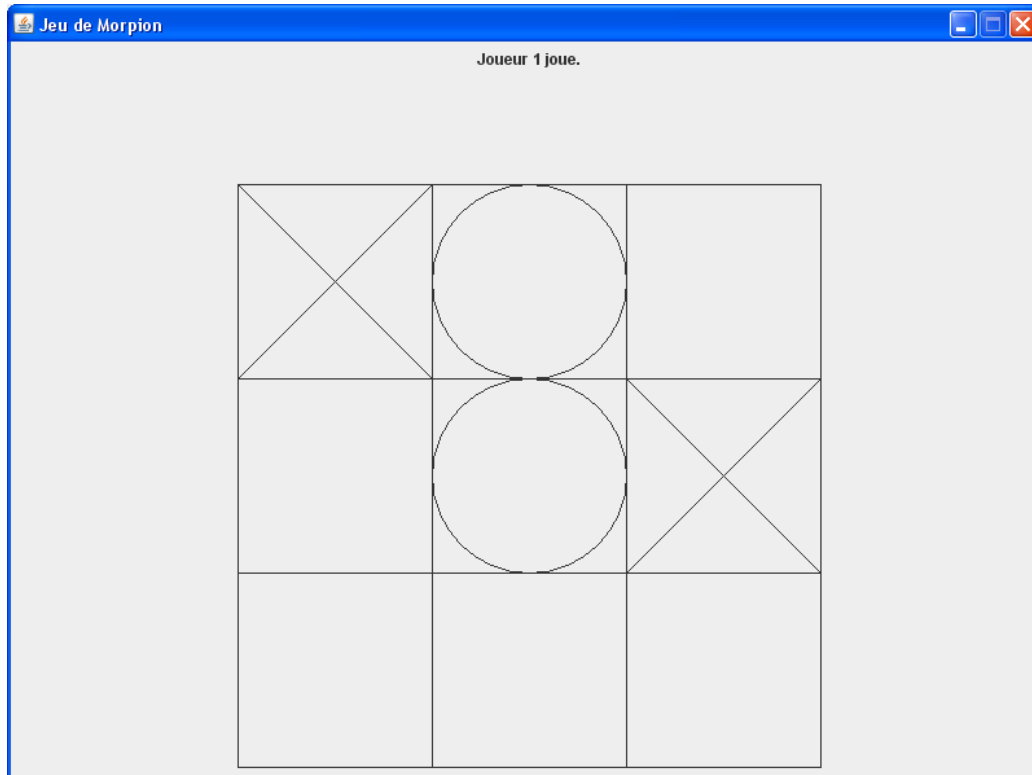
Annexes : manuel d'utilisation



Lorsque le programme est lancé, on voit quatre boutons :

- Le bouton « Nouvelle Partie » qui permet de lancer une nouvelle partie de morpion.
- Le bouton « Apprentissage rapide de l'IA » qui permet de lancer un grand nombre de parties entre deux ordinateurs, et ainsi alimenter leurs bases de connaissances.
- Le bouton « Options » qui permet de modifier un certain nombre de paramètres.
- Le bouton « Quitter » qui permet de quitter l'application.

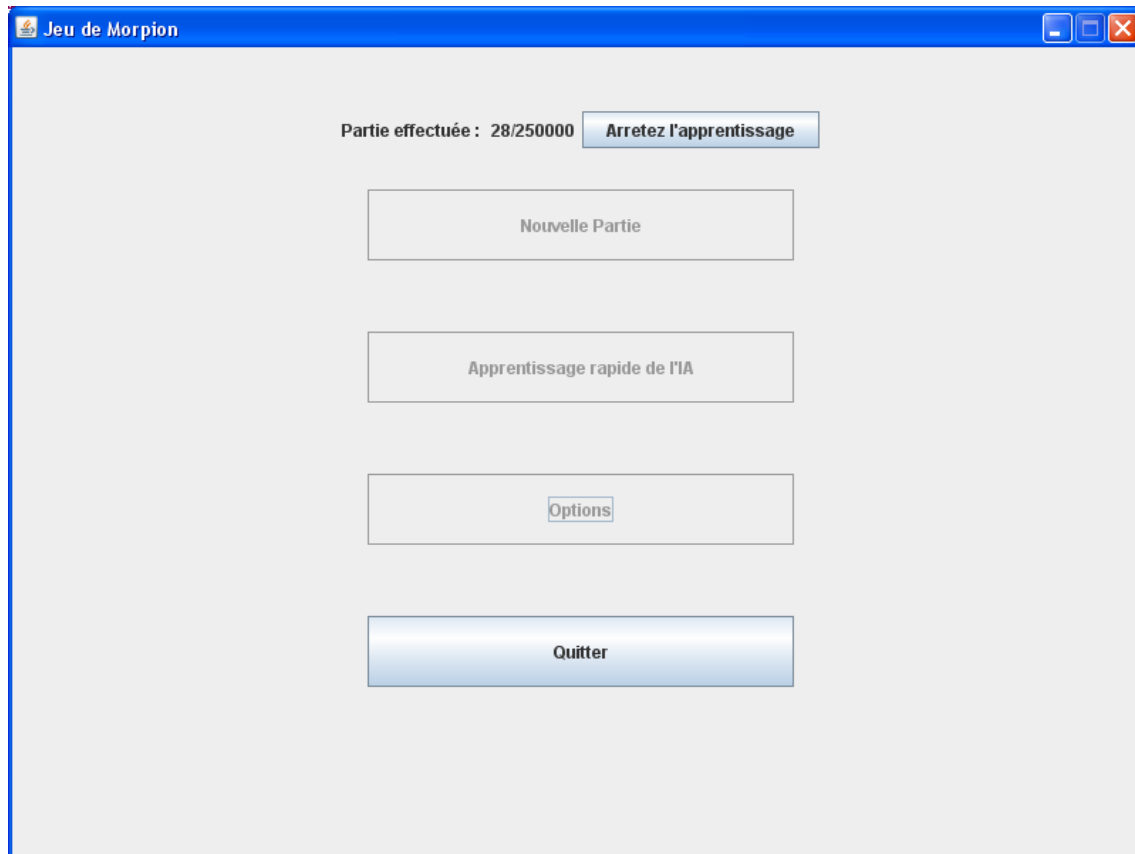
- Bouton « Nouvelle Partie » :



Apprentissage du jeu de morpion

Dans cette fenêtre, le joueur 1 et le joueur 2 jouent successivement jusqu'à la victoire d'un des deux joueurs ou par un match nul.

- Bouton « Apprentissage rapide de l'IA » :



Dans cette fenêtre, on voit le nombre de parties qui ont déjà été effectuées entre les deux ordinateurs et on peut arrêter l'apprentissage à tout moment en cliquant sur le bouton « Arrêtez l'apprentissage ».

- Bouton « Options » :



Dans cette fenêtre, on peut :

- Choisir si le joueur 1 est un joueur humain ou un joueur ordinateur.
- Choisir si le joueur 2 est un joueur humain ou un joueur ordinateur.
- Remettre à zéro l'apprentissage du joueur 1.
- Remettre à zéro l'apprentissage du joueur 2.
- Retourner au menu principal.

- Bouton « Quitter » :

Avec ce bouton, on peut quitter l'application.